ISI/SR-75-4
December 1975

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Inconsistency of a Single Data Value Over Time

Richard Bisbey II

Gerald Popek

Jim Carlstedt

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/SR-75-4	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Protection Errors in Operating Systems: Incansistency of a Single Data Value Over Time.	5. TYPE OF REPORT & PERIOD COVERED: Research Report. / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Richard Bisbey, II, Gerald Popek, Jim Carlstedt	8. CONTRACT OR GRANT NUMBER(s): DAHC15-72-C0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Informatian Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order #2223
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: December 1975 / 13. NUMBER OF PAGES: 20
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): ------	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

------

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Critical functian, inconsistent parameter, normalized representation, operating system security, protection policy, search process

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

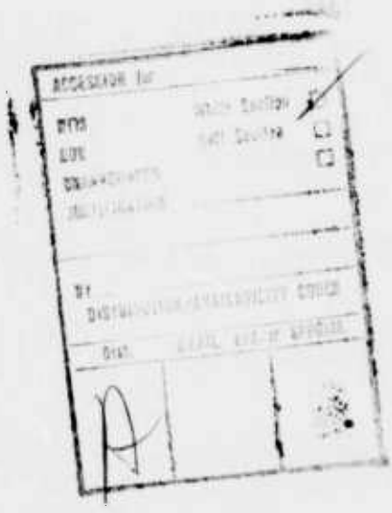
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

407 952

20. ABSTRACT

This report describes a pattern-based approach for finding a general class of computer operating system errors characterized by the inconsistency of a data value between pairs of references. A formal description of the error class is given, both as a protection policy being enforced and as a violation of that policy, i.e., an error statement. A particular subclass of the general error class is then examined, i.e., those errors in which the data type is a parameter. A formal specification of a procedure for finding instances of the subclass is given with examples of errors found using the procedure.

This work has been performed under Advanced Research Projects Agency Contract DAHC15 72 C 0308. It is part of a larger effort to provide securable operating systems in DOD environments.



ISI/SR-75-4
December 1975

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Inconsistency of a Single Data Value Over Time

Richard Bisbey II

Gerald Popek

Jim Carlstedt

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308. ARPA ORDER NO. 2223 PROGRAM CODE NO. 3D30 AND 3P10

ABSTRACT

This report describes a pattern-based approach for finding a general class of computer operating system errors characterized by the inconsistency of a data value between pairs of references. A formal description of the error class is given, both as a protection policy being enforced and as a violation of that policy, i.e., an error statement. A particular subclass of the general error class is then examined, i.e., those errors in which the data type is a parameter. A formal specification of a procedure for finding instances of the subclass is given with examples of errors found using the procedure.

This work has been performed under Advanced Research Projects Agency Contract DAHC15 72 C 0308. It is part of a larger effort to provide securable operating systems in DOD environments.

INCONSISTENCY OF A SINGLE DATA VALUE OVER TIME

PREFACE

This document is one in a series of related reports, each of which describes a specific type of security error found in current computer operating systems and presents techniques for finding errors of that type in a variety of systems (different versions, manufacturers, etc.). The series is based on a common methodology, i.e., the use of formalized error patterns to direct the search for errors corresponding to the patterns. The results are intended for use by persons responsible for the evaluation or enhancement of the security of existing operating system software. These studies will make it possible for individuals having no particular expertise in the field of operating system security to effectively carry out these tasks. Additionally, they can be utilized to influence the design of future systems or to reevaluate the protection mechanism of a system after it has undergone a release change. A more general description of the methodology can be found in [Carlstedt 75].

Like others in this series, this report is organized as follows. After an example error, the error type is presented in an abstract and formalized Policy/Error Statement, a representation developed during the analysis of errors and the derivation of error types carried out by ISI's Protection Analysis Project. The notation used in the formal description makes it easy to compare and classify error types. The technical terms have (necessarily) been given very precise meanings during the course of our work; definitions of these terms accompany the description of the pattern. The concise representation is accompanied by an informal description of the error type, including notes on its relations to other types.

Second, some of the primary "instantiations" of this pattern are described. An instantiation is the replacement of one or more relatively abstract terms in a pattern with one or more relatively concrete ones, resulting in a new pattern that is an instance of the former one. Instantiated patterns (may) fall into classes such that significantly different techniques and algorithms are applicable for finding errors described by the patterns of those classes. Each such class may be represented by a "subpattern."

The remaining sections of the report present techniques and algorithms useful for finding errors of a major subpattern.

The security error being discussed here is generic in nature and, like many errors, occurs many times in a number of different operating systems. Therefore, it is stated in system-independent terms. In order to automate the application of the pattern, both the pattern and the operating system must be described in common terms. However, the detailed representations of "target objects" contained in operating system X (Multics, for example) are likely to be different from those in system Y (OS/360, perhaps). In the two systems mentioned, X is written in a high-level language, while Y exists as assembly code. The instruction sequences in each system exhibiting the flaw may appear markedly

different in details despite the common thread. Thus to apply a pattern to a system, one of two things must occur: Either the pattern must be rerepresented in target level operating system terms ("instantiation") or the system must be rerepresented in the system-independent terms of the pattern ("normalization"), or some combination of both. In order to process each system with respect to a given error pattern, a different target instantiation/normalization in principle is necessary. Of course, since the work involves the same error type, the additional work will be modest. This paper describes one such target instantiation/normalization as a practical illustration of the underlying research. Other applications of the same pattern are merely engineering extensions of the method.

INTRODUCTION: AN EXAMPLE ERROR

Users communicate with operating systems by invoking supervisor subroutines (often called SVC's, MME's, UUO's, JSYS's, etc.) passing parameters describing both the operation and the object(s) on which the operation is to be performed. The supervisor procedure retrieves values from the parameters and possibly stores new values in the parameters before returning to the user. If the supervisor procedure fetches a parameter more than once or stores into a parameter and later fetches that same parameter, the expectation is that the value fetched is the same as the value previously fetched or stored. This may not be the case. Consider the following example procedure which performs the same function as the CNDIR JSYS in TENEX:

```
1.      connect:PROCEDURE (directory,password,code);
2.              CALL password_check (directory,password,code);
3.              IF code = 'ok' THEN
4.                      user_directory = directory;
5.              END;
```

Connect is a user-callable supervisor procedure allowing users to change their current file directory to a directory other than that which was specified at login. The procedure requires as parameters the name and password of the new working directory, and a variable in which a code indicating the result of the request is returned to the caller. When called, the connect procedure invokes the supervisor procedure "password_check" to verify the correctness of the directory/password pair. If the pair is correct, the working directory is changed and the status code 'ok' is returned to the caller. If the directory/password pair is incorrect, no directory change is made and an error code is returned.

The connect procedure assumes that the value of "directory" referenced in line 4 is the same as that checked in the procedure call in line 2. Similarly, the procedure assumes that the value of "code" referenced in line 3 is the same as that stored by the procedure "password_check" in line 2. In both cases, if the value is changed, an exploitable security flaw results. For example, suppose a user wished to connect to directory SMITH, but only knew the password for directory JONES. One way would be to

1. Call connect with parameters JONES/Jones's password.
2. After the execution of line 2 but before the execution of line 4 in the connect program, change the "directory" parameter to SMITH.

Another way, not requiring knowledge of any password, would be to

1. Call connect with parameters SMITH/any string.
2. After the execution of line 2 but before the execution of line 3 in the connect program, change the "code" parameter to 'ok'.

In both cases, the entire password check would be invalidated and the user could connect to any directory regardless of password.

Since the above errors are crucially dependent on the ability of the user to modify a parameter, it is worthwhile to enumerate some of the possible ways this could occur.

1. If the parameter in question is passed by name or reference, the cell containing the value remains in the caller's address space throughout the execution of the called code. The value of the cell could be changed by any asynchronous process having write access to the caller's address space. Such processes include user and system procedure code and input/output.
2. Again, if the parameter is passed by name or reference, the calling procedure could arrange that input and output parameters occupy the same physical storage. Thus, when the called procedure stores into the output parameter, it is also storing into, and modifying, the value of the input parameter.

The above are examples of direct changes to the value. The name/cell binding of the cell could also be changed.

3. Many machines have a hardware feature whereby the address contained in an indirect addressing word is incremented each time it is referenced. The caller could pass a parameter specifying such an indirect word. Each time the called procedure references the parameter, it references a different cell, which could contain a different value.

THE GENERAL ERROR TYPE

The general error type addressed in this paper concerns the class of errors in which a data value is rendered inconsistent between two operations. The errors in the connect procedure above are specific instances of this error type, as are IBM's TOCTTOU Errors [McPhee 74].

For an operating system to be secure, one must insure that the data which the system uses to make security and protection-related decisions is correct. By "correct" we mean that the value of the data must be within a range admissible to the protection operations, and it must be consistent from one usage to the next. Incorrect data can result in actions being taken by the system which allow the protection mechanism to be violated and the system compromised.

The notion of consistency of value between two operations can be captured formally by the following Protection Policy statement:

POLICY:
(B,M,X) => for some operation L occurring before M, either
[for operation L which does not modify Value(X),
Value(X) before L = Value(X) before M], or
Value(X) after L = Value(X) before M.

More intuitively stated, process B (which presumably performs some critical function) can perform operation M on variable X only if the value of X at the time operation M is performed is equal to the value of X either before or after some operation L which occurs before M.

As mentioned in the preceding example, there are two possible ways for a value to be rendered inconsistent. First, the value of the cell may be changed directly by storing a new value in the cell. This could be done either by the executing process or by some other process in the system. The resulting inconsistent (possibly out-of-range) value invalidates the conditions under which M is allowed to operate, resulting in an error. Second, the value may be changed by altering the name/cell binding used to access the cell. When a program accesses a cell, the name used in the access is interpreted by the addressing mechanism which translates the name into the physical location of the cell. If this binding of the name of the cell and its physical location is changed by substituting a different cell for the name X, the effect can be equivalent to changing the value of the cell X. Examples of objects which affect name/cell binding include indirect addresses, addressing registers, and memory maps.

The Error Statement corresponding to the above Policy Statement is

ERROR:
B:M(X) and for some operation L occurring before M,
[for operation L which does not modify Value(X),
Value (X) before L NOT = Value (X) before M], and
Value (X) after L NOT = Value (X) before M.

Informally stated, process B performs operation M on variable X and the value of X at the time operation M is not performed is equal to the value of X either before or after some operation L which occurs before M.

INSTANTIATION: INCONSISTENCY OF PARAMETER VALUES

This instantiation investigates the application of the above Policy and Error Statements to interdomain procedure calls with parameters. The parameter corresponds to the variable X in the above Policy and Error statements.

STATICIZED PATTERN

To find instances of the above error in code, a pattern is formed using the Error Statement. The Error Statement requires the existence of two operations, both of which refer to a common variable X. The first operation, L, either fetches the value of the variable or generates a new value. The second operation, M, fetches the value of the variable. Other information contained in the Error Statement includes the fact that L occurs before M and that M performs some critical function. These statements give rise to the following pattern elements:

1. Find an operation L which either fetches or stores into a cell X.
2. Find an operation M which fetches cell X.
3. Operation M is critical.
4. Operation L occurs before operation M.

For this particular error, X is instantiated to a parameter, and thus the following additional pattern element is derived:

5. Find a procedure B which is inter-domain callable by user procedures and which accepts a parameter X.

The above five pattern elements, when intersected, will find all instances of the given error. The salient features of the elements are:

a) procedure inter-domain callable by user procedures
b) parameter
c) code that fetches or stores the value of a given variable
d) code that fetches a given variable
e) the "before" relation
f) critical function.

The above features constitute an appropriate set of objects for a comparison language; an operating system could be mapped into these objects (i.e., normalized) and a simple comparison algorithm could be written to find sequences of the desired objects. Not all of the features, however, are directly represented in static procedure code. The "before" relation of pattern element 4 is not a discrete object in procedure code. It is calculated by interpreting flow of control operators in the static code. In principle, the normalization process could calculate the relation for a symbol and all other symbols in the code and output all instances of the relation as discrete objects. In order to simplify the normalization process, the above calculation will not be made; instead, flow of control operators will be included in the normalized representation. An interpretive routine will

then be used during comparison to calculate the relation for particular cases. Another method of simplifying both the normalization and search process is to relax the pattern by dropping pattern elements 3, the "criticality" condition. As a result of this relaxation, detected instances of the pattern must be examined manually to determine whether they are in fact protection errors.

The normalization procedure is now the following:

1. Filter out everything except procedures which are inter-domain callable by users.
2. Of these, identify those with parameters.
3. For each parameter, identify and output all instructions or statements which involve store or fetch operations on the parameter.
4. Identify and output all instructions or statements which contain flow of control operators.

The search process is to recognize, for each parameter, executable sequences of store or fetch operations followed by a fetch operation. The matches are thereafter examined manually to determine if the second operation was in fact critical.

NORMALIZATION AND SEARCH NOTES

The following is a list of special considerations for normalizing and searching the resulting normalized representation.

1. Care must be exercised in collecting flow of control operators from the original program text and in interpreting them in the search process so that multiple execution of a single reference such as in a repetitive loop is detected. As an example, consider the following:

```
X:PROCEDURE (i);
        ...
        DO WHILE (...);
                ...
                ... = i;
                ...
        END;
        ...
END;
```

   Even though the parameter "i" appears only once in the static code, the reference may be executed multiple times, constituting a possible error.
2. It may not always be possible to determine the usage of a variable from local context, e.g., parameters used as arguments in function or procedure calls. In these cases, the worst-case assumption of both fetch and store should be used.
3. Care must be exercised in identifying statements which involve parameters. The normalization process must detect not only direct usage of the parameter, but also indirect usage by overlay or redefinition of physical storage or names, and by pointer variables and pointer addressing.

4. Special attention should be given to parameters in which pointers or addresses are passed. In such cases, even though the parameter may be passed by value, the effect of passing the address is usually equivalent to a reference parameter.

EXAMPLE APPLICATION

The inconsistent parameter instantiation of the above error type was applied to 47 procedures from the Multics operating system. The procedures were selected from the the user-callable gate "hcs_". A program was written to automate a large portion of the normalization task. Specifically, the program processed each procedure, identifying and outputting by parameter a list of statements in which parameter usage occurred. The access type for each usage was then identified as either "fetch," "store," or "fetch/store." This list was then searched manually for sequences of either fetch, store, or fetch/store followed by fetch. All matching pairs were then examined to determine if the second fetch was critical.

Of the 47 procedures examined, seven were observed to have one or more errors; five other procedures had matches for which "criticality" of the second fetch could not be determined due to lack of system documentation.*

The errors found are highlighted by the following three examples (where the original parameter is denoted by italics). In example 1, the program obtains the address of the parameter and stores it in a local variable. The procedure next copies the parameter value to a local variable and tests the value for range. When the procedure later uses the parameter in a calculation, it references the original value using the stored pointer. This not only invalidates the range checking code, but also hides the final erroneous fetch because of the pointer usage. In example 2, the program fetches the parameter many times. Different values at each fetch cause the procedure to perform erroneously. In example 3, the program checks the value of the parameter before using it in a critical procedure call. The value checked, however, may be different than that used in the procedure call.

---

* It is important that the reader draw no conclusion from this statement regarding the security of MULTICS relative to other systems other than the fact that this error pattern has been applied to MULTICS and the observed errors repaired.

```
1 /* ******************************************************
2    *                                                    *
3    *                                                    *
4    * Copyright (c) 1972 by Massachusetts Institute of   *
5    * Technology and Honeywell Information Systems, Inc. *
6    *                                                    *
7    *                                                    *
8    ****************************************************** */
 ...
33         dcl 1 channel_name aligned based(chn_ptr), /* structure of channel name */
34                 2 rel_ptr bit(18) unaligned, /* points to channel */
35                 2 ring bit(3) unaligned, /* ring of channel */
36                 2 pad bit(15) unaligned,
37                 2 channel_index fixed bin(35); /* index of special channel */
 ...
54 ipc_sf_block:      entry(channel,code);
55
56         esw = 3; /* indicate entry */
57         val_ring = level$get(); /* get validation level */
58         chn_ptr = addr(channel); /* get pointer to channel name */
59         ev_ring = fixed(channel_name.ring,3); /* get channel ring */
60         cindex = channel_name.channel_index; /* copy index */
61         if ev_ring = val_ring then do; /* rings don't match */
62                 code = error_table_$wrong_channel_ring;
63                 return;
64                 end;
65         if channel_name.rel_ptr then do; /* not a special channel */
66                 code = error_table_$invalid_channel;
67                 return;
68                 end;
69         if cindex < 1 | cindex > tc_data$max_channels then do;
70                         /* cannot be special channel */
71                 code = error_table_$invalid_channel;
72                 return;
73                 end;
 ...
111                         if return_sw = 1 then code =
 error_table_$ips_has_occurred; /* remember that ips */
112                          else code = 0;
113                         if substr(pds$events_pending
  pds$event_masks(val_ring),cindex,1) then do; /* we have wakeup */
114
 substr(pds$events_pending,channel_name.channel_index,1) = "0"b;
115                                 return_sw = 1; /* we can return */
116                                 code = 0;
117                                 end;
 ...
```

Example 1

```
1 /* ****************************************************
2    *                                                  *
3    *                                                  *
4    * Copyright (c) 1972 by Massachusetts Institute of *
5    * Technology and Honeywell Information Systems, Inc.*
6    *                                                  *
7    *                                                  *
8    **************************************************** */
 ...
11 tty_write: proc (twx, readp, offset, nelem, nelemt, state, ercode); /* tty output
 conversion */
 ...
22 dcl  ercode fixed bin (35), nelem, nelemt, offset, state, twx, col, tidx, tmp1, tmp2, i, ini,
 inmax) fixed bin (18),
 ...
180          ini = offset;                              /* in char index */
181          inmax = ini+nelem;                           /* input index limit */
182
183          if gterm then if graphic then go to gmode;       /* if in graphic mode go there
 immediately */
184
185          if dev_info.hs_dev then go to hs_write;          /* if high speed channel do
 it there */
186
187 loop:     if ini >= inmax                             /* if done */
188               then do;
189               donesw = "1"b;                           /* we are done except for
 updating carrage */
190               go to asblack;
191          end;
192
193          if iaend_frame                               /* if frame full */
194               then go to full;
195
196          if ini-offset > 4095 then do;                  /* write max of 4095 on one call
 to prevent */
197                                                      /* a user from hogging hardcore and
 processor */
198               ercode = 3000005;                        /* let user know we are not done
 with write yet */
199               donesw = "1"b;                           /* done for now */
200               go to asblack;                          /* start writing what we've got */
201          end;
202
 ...
```

Example 2

```
1 /* ******************************************************
2    *                                                    *
3    *                                                    *
4    * Copyright (c) 1972 by Massachusetts Institute of   *
5    * Technology and Honeywell Information Systems, Inc. *
6    *                                                    *
7    *                                                    *
8    ****************************************************** */
9
10 stop_process: proc(process_id);
11
12 /* Procedure used by a process to put itself into "stopped" state.
13    Called by process-overseer on new_proc and logout.
14
15    Converted to PL/I, C Garman, 3 Feb 1971.
16
17    */
18
19 dcl process_id bit(36) aligned;
20
21 dcl pds$process_id bit(36) aligned ext,
22     istate fixed bin;
23
24 dcl pxss$stop entry(bit(36) aligned, fixed bin);
25
26          if pds$process_id ¬= process_id              /* See if proper process */
27          then go to return;
28
29 /* If super-stop ever wanted, from hphcs_, delete this line.
30
31 stop_process_irmi: entry(process_id);                  /* "possible hphcs_
 entry" */
32
33          call pxss$stop(process_id, istate);
34
35 return:
36 end stop_process;
```

Example 3

REFERENCES

[Carlstedt 75] Carlstedt, J., Bisbey, R., Popek, G., ***Pattern Directed Protection Evaluation,*** USC Information Sciences Institute, ISI/RR-75-31, June 1975.

[McPhee 74] McPhee, W. S., ***Operating System Integrity in OS/VS2,*** IBM Systems Journal, Vol. 13, No. 3, 1974, pp. 230-252.